THE UNIVERSITY of EDINBURGH
informatics



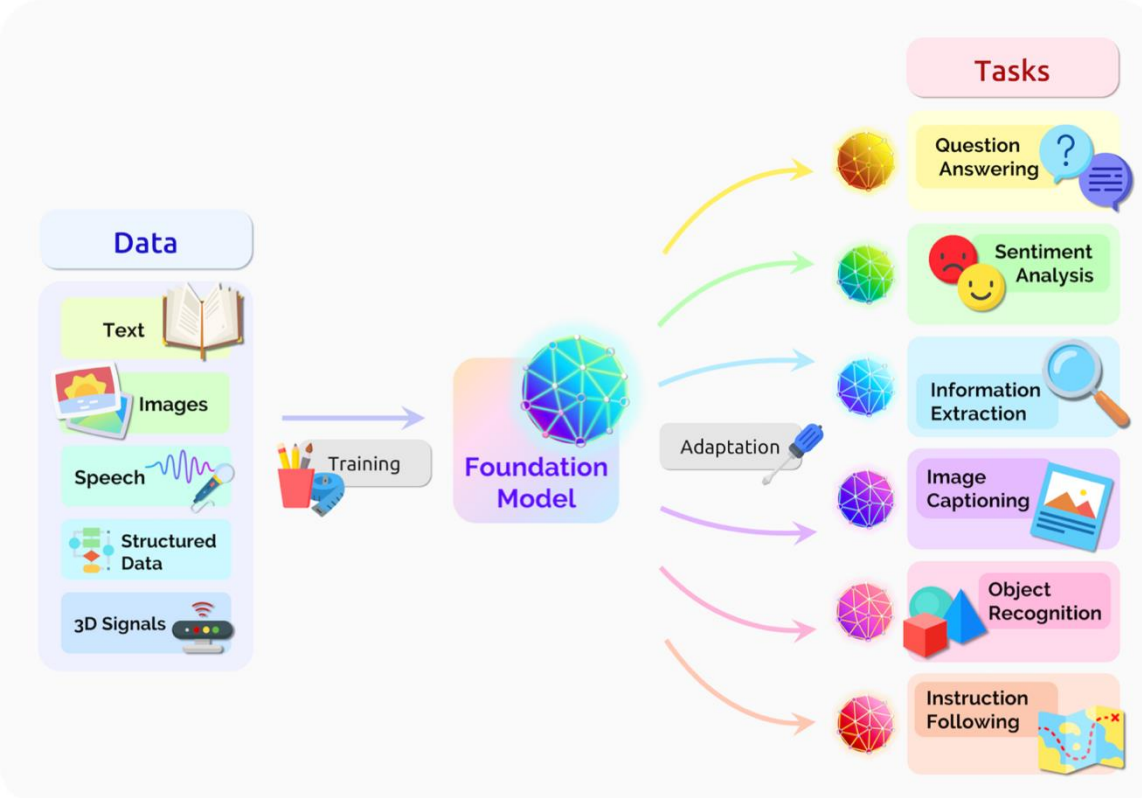
Design Principles of Training Systems

Luo Mai

University of Edinburgh



Large AI Models



LLM Models:

- GPT-3, GPT-4, GPT-5
- DeepSeek-V3 / R1 (MoE)
- Gemini 3 Pro

Multimodal Models:

- Diffusion models + Transformers
- Omni models

[1] On the Opportunities and Risks of Foundation Models, 2021

Compute demand grows much faster than hardware



Gap to be filled by **Distributed AI Systems**

Figure: OpenAI, "AI and Compute" (2018).

The trend has continued: frontier training compute grew about **4–5× per year** from 2010 to 2024 (Epoch AI) — far faster than chips improve.

Common reasons for distributed ML systems

- Performance
  - Reducing the time to complete a data epoch
- Memory wall
- Economy
  - Multiple commodity servers, instead of a single expensive high-end server
- Hardware failure tolerance

How much compute does training need?

Training FLOPs ≈ 6 × N × D

N = number of parameters D = number of training tokens

- **Why 6?** Each parameter costs about 2 FLOPs per token in the forward pass and about 4 in the backward pass.
- **MFU** (Model FLOPs Utilisation) = useful model FLOPs per second ÷ the hardware's peak FLOPs per second.
- Well-tuned large runs reach about 35–45% MFU in BF16. The rest is lost to communication, memory stalls, pipeline bubbles and failures.

Worked example: Llama 3.1 405B [1]

N = 405 B parameters, D = 15.6 T tokens

6ND ≈ 3.8 × 10^{25} FLOPs

H100 peak (BF16): 989 TFLOP/s → about 400 TFLOP/s at 40% MFU

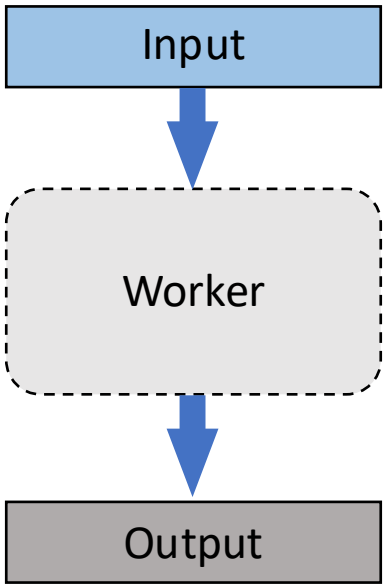
~3,000 years
on one GPU

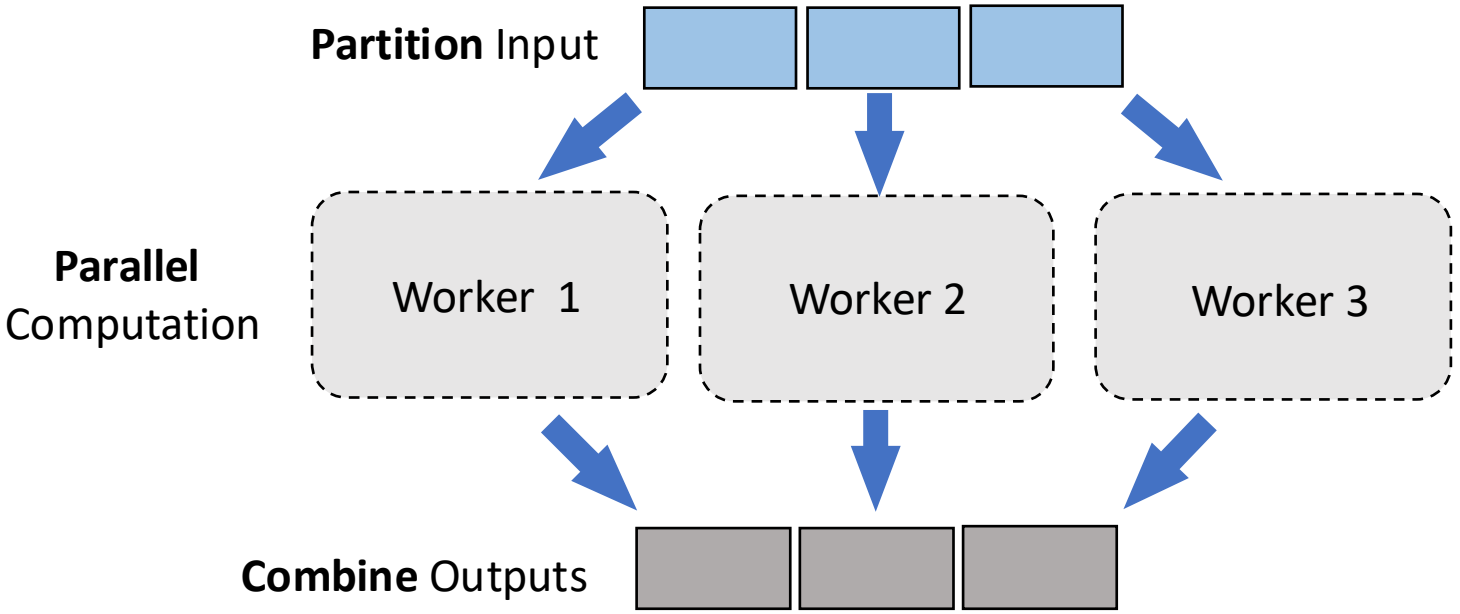
~68 days
on 16,384 GPUs (perfect scaling)

Parallelism turns millennia into months. Here, each 1% of MFU lost costs about 0.7 M GPU-hours.

[1] Grattafiori et al., The Llama 3 Herd of Models, 2024.

Divide-and-Conquer

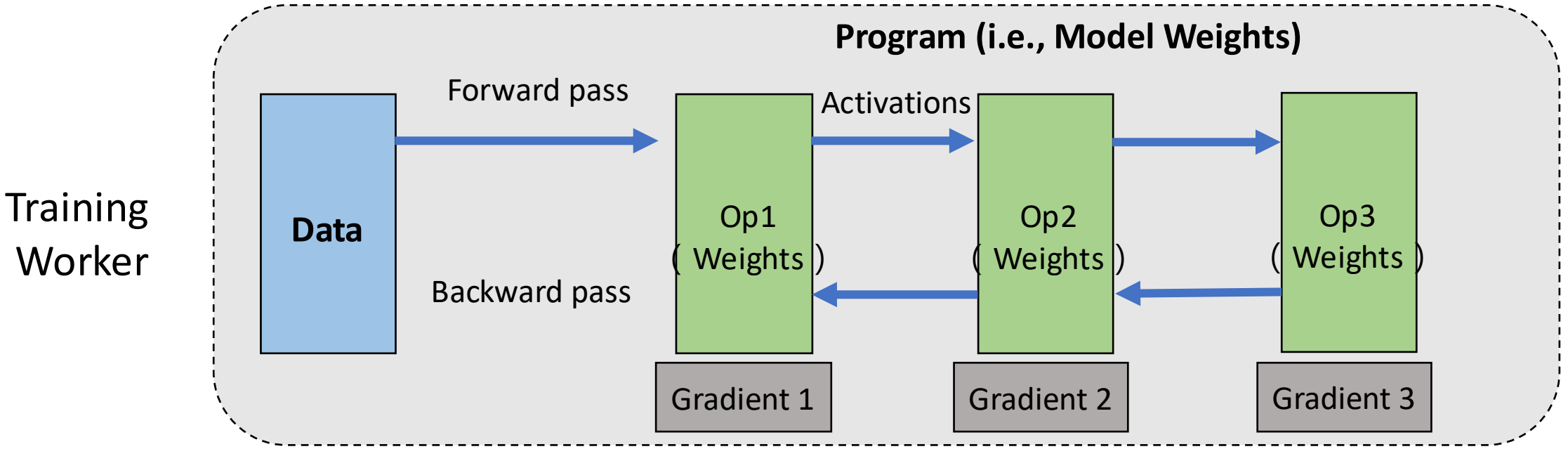




(a) Single-worker Execution

(b) Distributed Multi-worker Execution

Basic execution model



Workers must have sufficient memory to store data, weights, activations & gradients

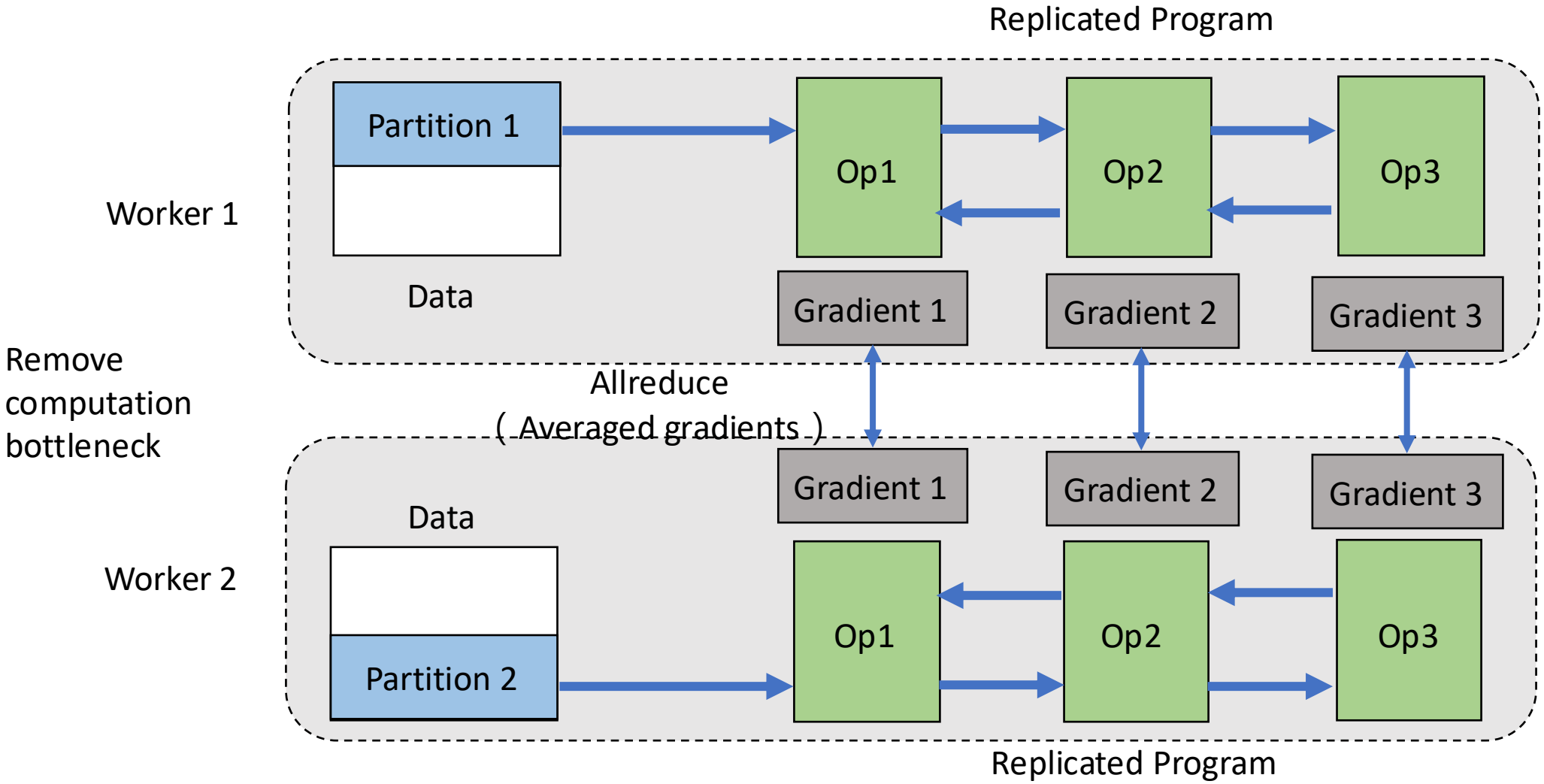
- Otherwise, you get Out-Of-Memory (OOM) exception

Parallel training methods

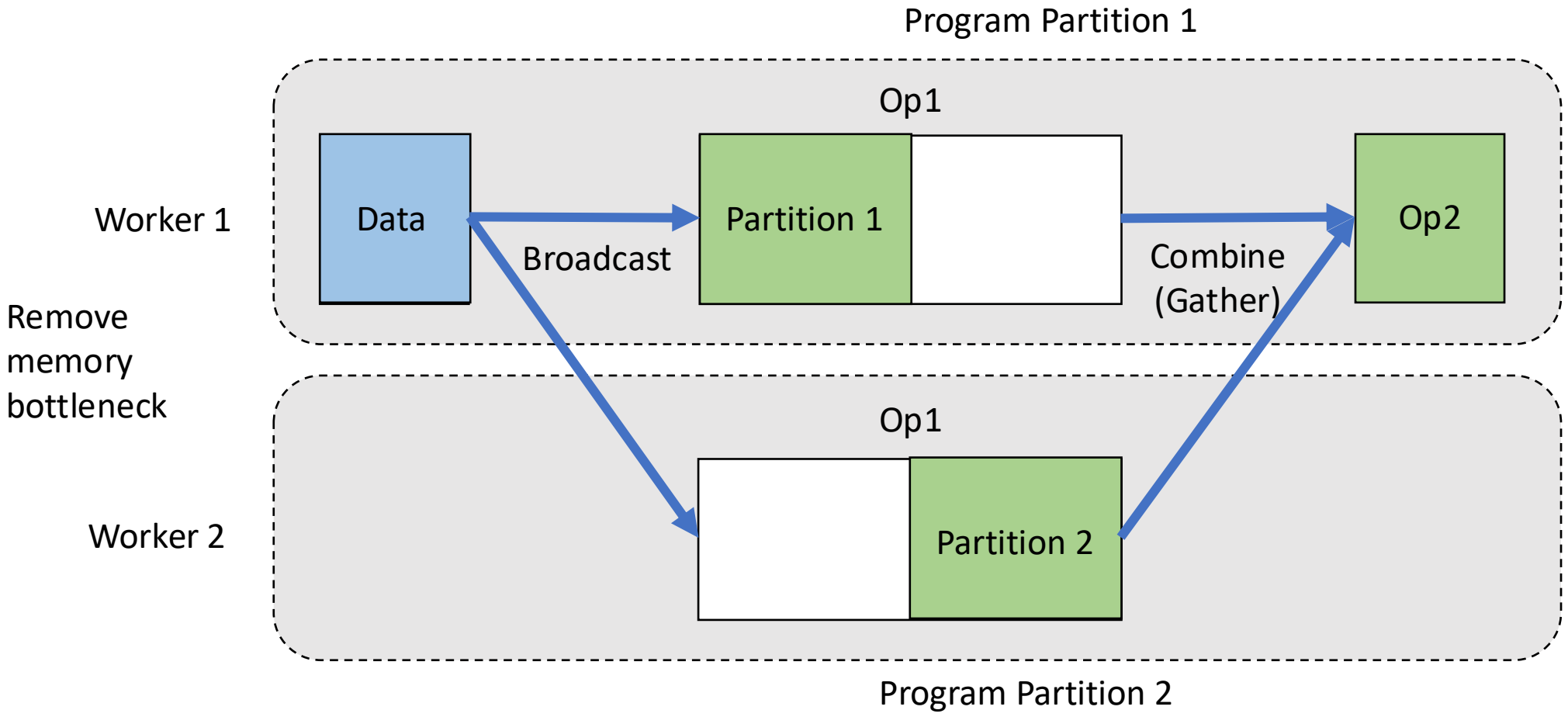
	Single Data	Multiple Data
Single Program	Single-Program-Single-Data (SPSD) **Single Worker Training**	Single-Program-Multiple-Data (SPMD) **Data Parallel Training**
Multiple Program	Multiple-Program-Single-Data (MPSD) **Model Parallel Training**	Multiple-Program-Multiple-Data (MPMD) **Hybrid Parallel Training**

Data parallel training

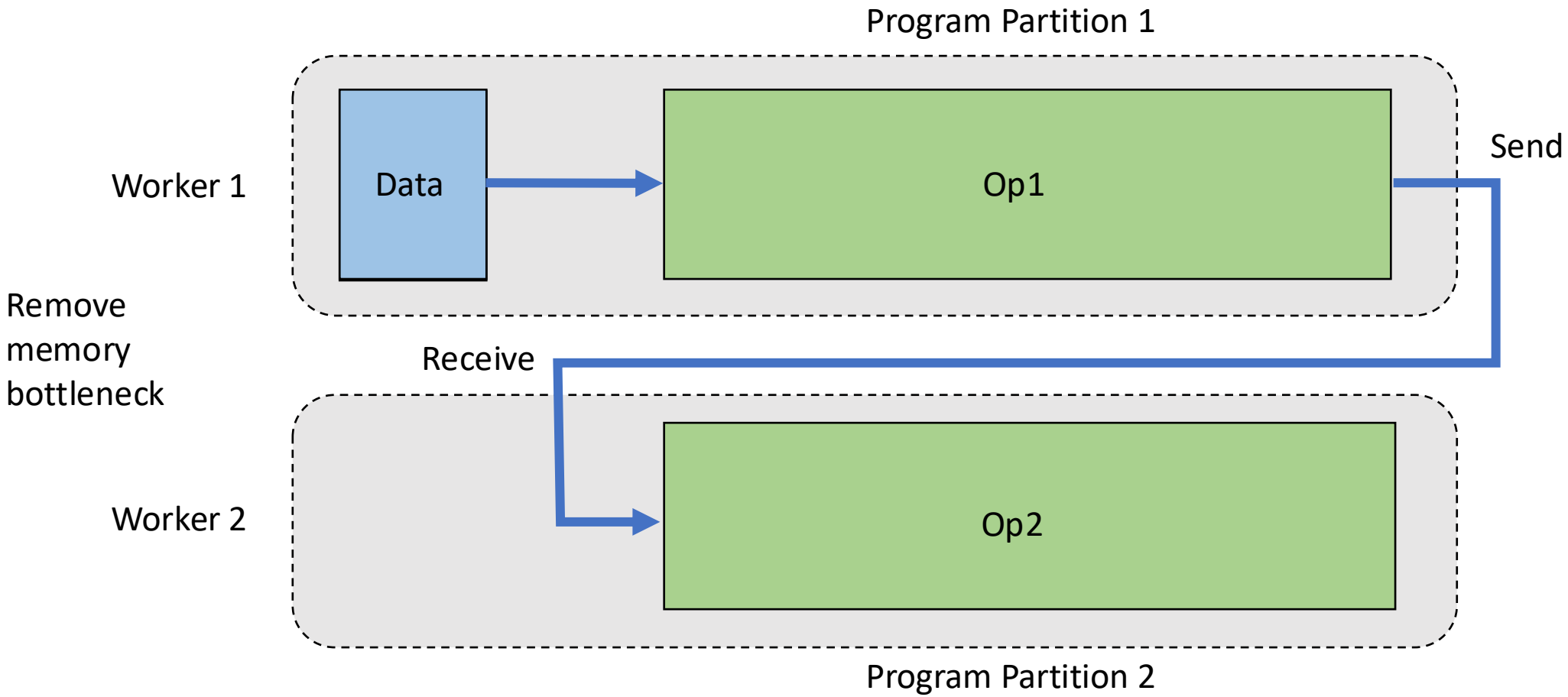
Parameters replicated; activations local; gradients synchronized.



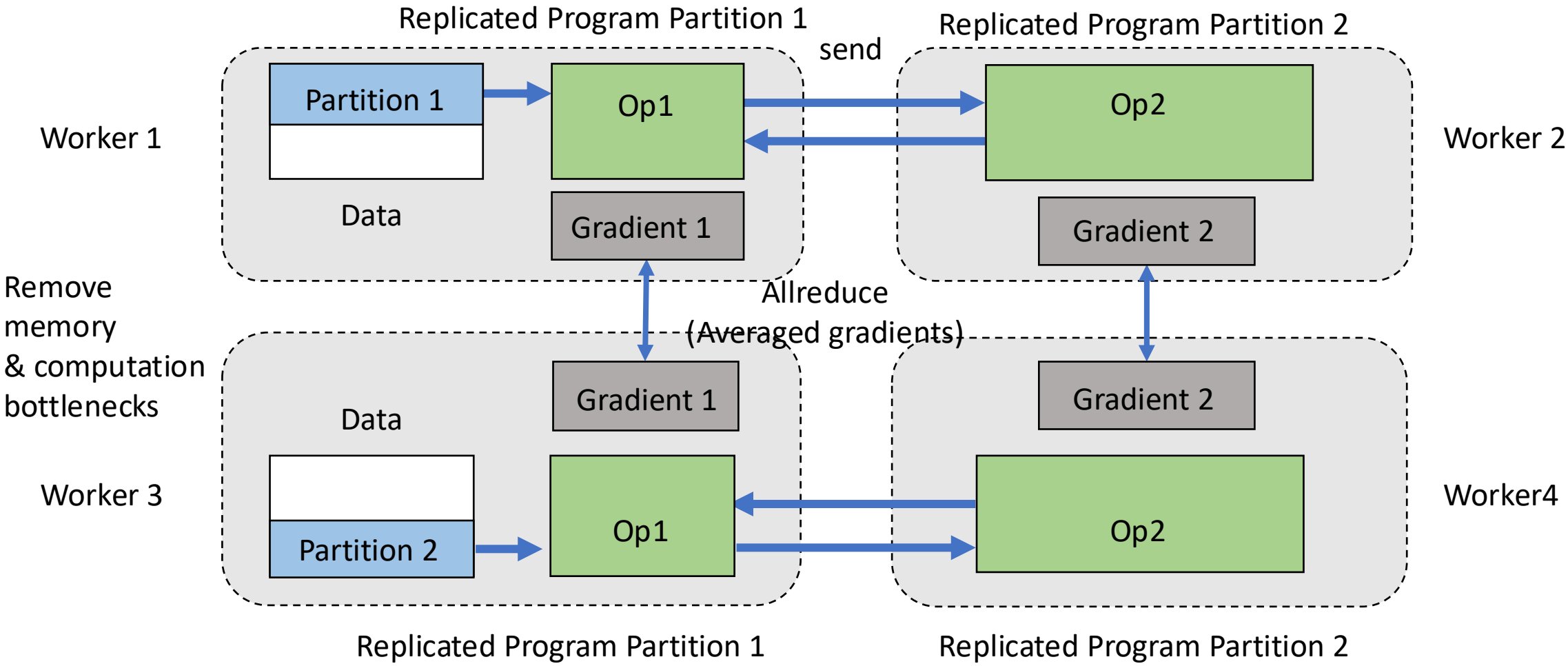
Model parallel training: Intra-operator



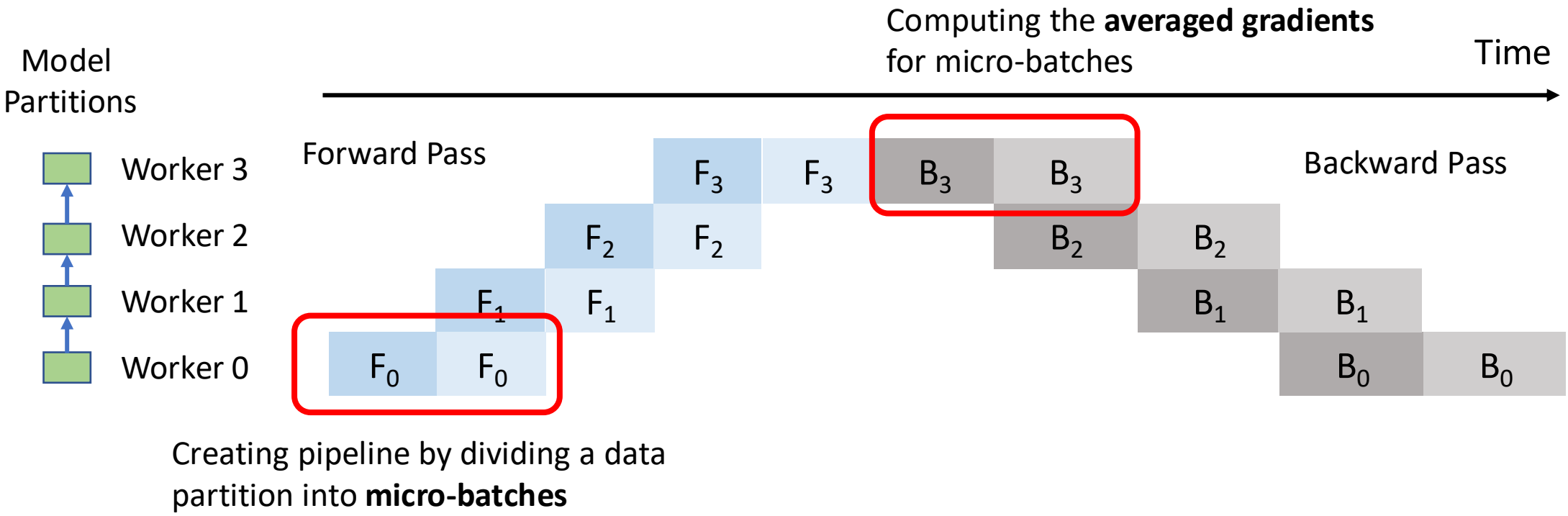
Model parallel training: Inter-operator



Hybrid parallel training



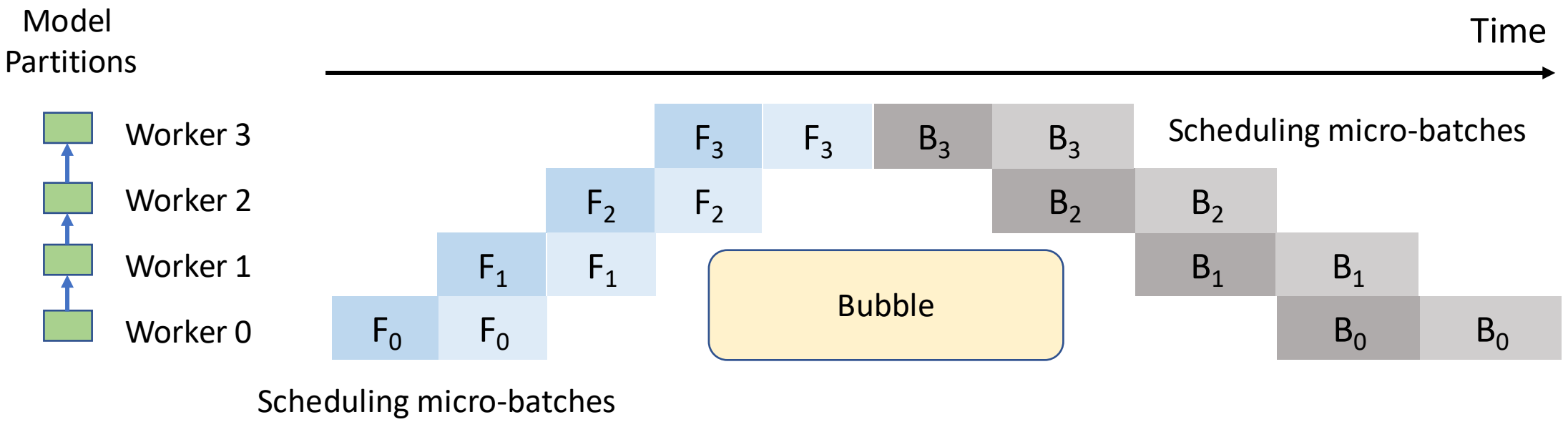
Pipeline parallelism



Creating pipeline by dividing a data partition into **micro-batches**

Activation checkpointing is a memory-saving technique used in **pipeline parallelism** to reduce GPU memory usage during training. Instead of storing all activations from forward passes for backpropagation, **only selected checkpoints are saved**, and the rest are recomputed during the backward pass.

Optimising micro-batch size



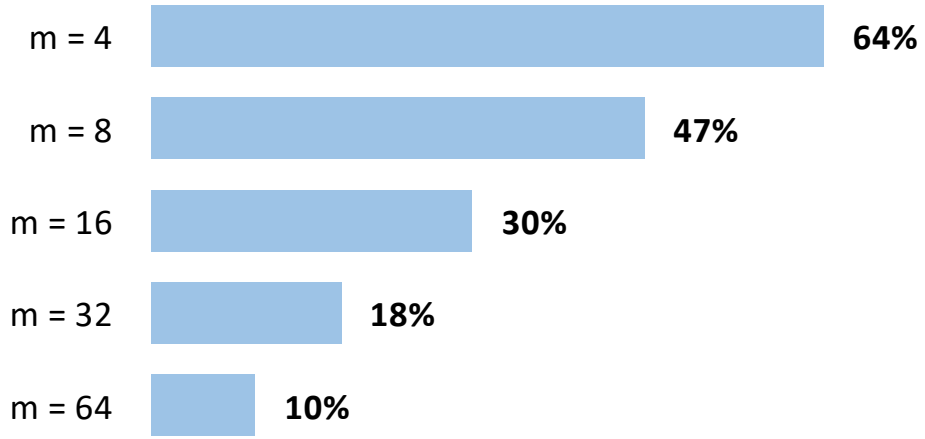
- **Small micro-batch** reduces bubble size; but incur large micro-batch scheduling overheads
- **Large micro-batch** incurs large bubble; but come with small micro-batch scheduling overheads
- Optimal micro-batch size must **balance bubble size and scheduling overheads**

Better pipeline schedules

Bubble ≈ (p – 1) / (m + p – 1)

share of step time where GPUs sit idle
p = pipeline stages, m = micro-batches per step

Example: p = 8 stages



More micro-batches shrink the bubble, but each micro-batch gets smaller and less efficient.

- **GPipe (2019):** run all forwards, then all backwards. Must keep activations for all m micro-batches.
- **1F1B (PipeDream, 2019–21):** after a warm-up, alternate one forward and one backward. Same bubble, but activations for at most p micro-batches.
- **Interleaved 1F1B (Megatron-LM, 2021):** each GPU holds v smaller model chunks. The bubble shrinks by about v×, at the cost of more communication.
- **Zero Bubble (2024):** split backward into input-gradient and weight-gradient parts. Weight-gradient work fills the bubbles.
- **DualPipe (DeepSeek-V3, 2024):** feed micro-batches from both ends of the pipeline and overlap MoE communication with compute.

Huang et al., GPipe, 2019 · Narayanan et al., PipeDream, 2019 / 2021 · Narayanan et al., Megatron-LM, SC 2021 · Qi et al., Zero Bubble Pipeline Parallelism, ICLR 2024

Case Study: Mixture of Expert Models

In Mixture-of-Expert (MoE) models, each FFN layer in the Transformer is replaced with an MoE layer, consisting of a gate network (often called as router) and multiple experts.

Key reasons for using MoEs:

- **Computational efficiency** (activating only a subset of experts per input, reducing FLOPs)
- **Specialization** (allowing different experts to focus on distinct data patterns for better performance)

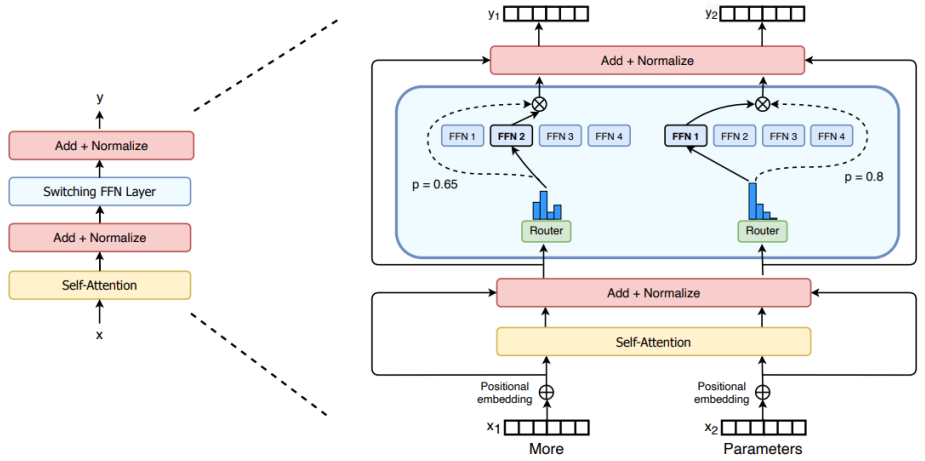
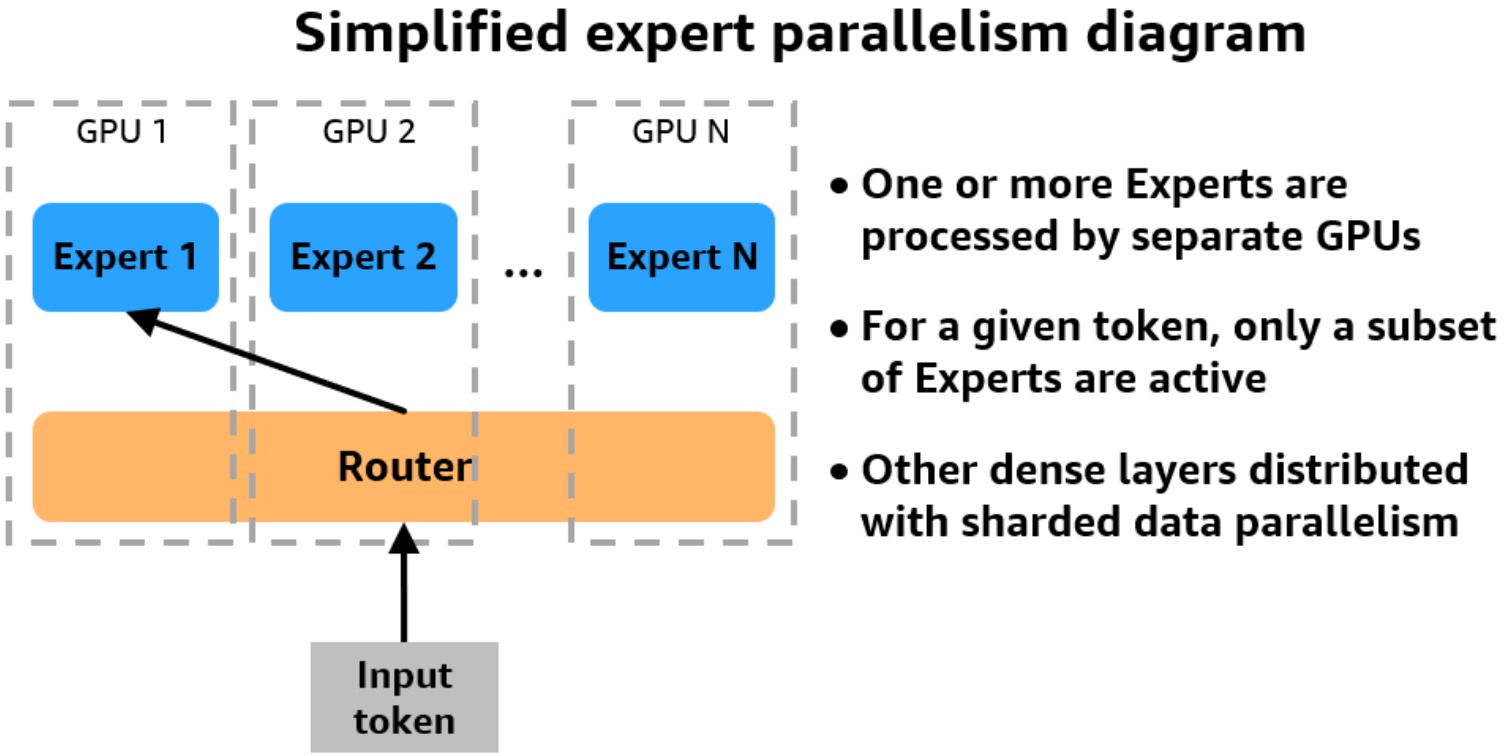


Figure 2: Illustration of a Switch Transformer encoder block. We replace the dense feed forward network (FFN) layer present in the Transformer with a sparse Switch FFN layer (light blue). The layer operates independently on the tokens in the sequence. We diagram two tokens (x_1 = "More" and x_2 = "Parameters" below) being routed (solid lines) across four FFN experts, where the router independently routes each token. The switch FFN layer returns the output of the selected FFN multiplied by the router gate value (dotted-line).

MoE layer from the [Switch Transformers paper](https://arxiv.org/abs/2101.03961)

Expert Parallelism

As MoE models grow too large for a single GPU, expert parallelism distributes experts across GPUs, enabling efficient model parallelism and better performance.



[1] https://aws.amazon.com/blogs/machine-learning/accelerate-mixtral-8x7b-pre-training-with-expert-parallelism-on-amazon-sagemaker/

Why expert parallelism is hard



Each MoE layer runs two all-to-alls in the forward pass and two more in the backward pass.

1. All-to-all traffic

Each token's hidden state travels to the GPUs that hold its experts, and back again. Across nodes, this can dominate the step time.

→ Keep EP inside the NVLink domain when possible.

2. Load imbalance

Popular experts receive more tokens. Their GPUs become stragglers, or extra tokens are dropped.

→ Balance loss, capacity limits, or bias-based balancing (DeepSeek-V3).

3. Hiding communication

All-to-all sits on the critical path: experts cannot start until tokens arrive.

→ Overlap it with another micro-batch's compute (DualPipe) and use optimised kernels (DeepEP, 2025).

Example: DeepSeek-V3 has 671 B parameters but only 37 B are active per token (256 routed experts, 8 chosen per token).

Activation Partitioning / Parallelism

Sequence Parallelism (SP)

What it is

- Sequence Parallelism extends **tensor (intra-layer) parallelism** by **sharding activations along the sequence dimension** for selected Transformer sublayers.

What is partitioned

- Activations of **specific layers** (typically layer normalization, dropout, and elementwise operations) are partitioned across GPUs.

Why it is used

- Reduces **activation memory** and balances computation in layers that are **not naturally tensor-parallel**, especially when tensor parallelism is enabled.

Context Parallelism (CP)

What it is

- Context Parallelism partitions the **entire sequence dimension** of the input activations across GPUs for **long-context Transformer training**.

What is partitioned

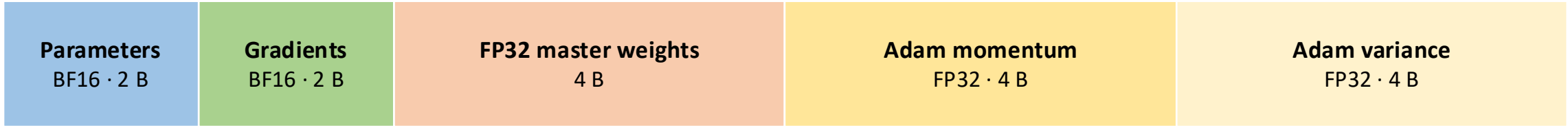
- Activations of **all Transformer layers** are partitioned by token (context position).

Why it is used

- Enables training with **very long sequences**, where attention and activation memory scale with sequence length.

Where does the memory go?

Model states per parameter, for mixed-precision training with the Adam optimizer: **16 bytes in total**



Example: GPT-3 (175 B parameters)

- 175 B × 16 bytes ≈ **2.8 TB** of model states
- At least 35 H100s (80 GB each) just to hold them
- Plain data parallelism keeps a full copy on every GPU

Activations come on top

- Grow with batch size × sequence length × hidden size × layers
- Often larger than model states for long sequences
- Cut by activation checkpointing (recompute) and SP / CP (shard)

ZeRO (next slide) removes the redundant copies of these 16 bytes across data-parallel GPUs.

[1] Rajbhandari et al., ZeRO: Memory Optimizations Toward Training Trillion Parameter Models, SC 2020.

Case Study: ZeRO (Zero Redundancy Optimizer)

ZeRO **reduces memory usage** by **sharding** optimizer states, gradients, and parameters across GPUs instead of replicating them.

Stage	What is sharded	Memory per GPU	175 B, 64 GPUs
Plain data parallel	Nothing (full copy everywhere)	16Ψ	2.8 TB
ZeRO-1	Optimizer states (12 of the 16 bytes)	4Ψ + 12Ψ/N	≈ 733 GB
ZeRO-2	+ Gradients	2Ψ + 14Ψ/N	≈ 388 GB
ZeRO-3	+ Parameters (everything)	16Ψ/N	≈ 44 GB

Ψ = number of parameters, N = number of data-parallel GPUs. Bytes per parameter follow the 16-byte breakdown on the previous slide.

The price: ZeRO-3 all-gathers each layer's parameters before using them, in both forward and backward. This costs about 1.5× the communication of plain data parallelism.

Case Study: ZeRO-Offload

What it is

- A ZeRO variant that **offloads optimizer states (and related computation)** from GPU memory to **CPU memory**.

Why it exists

- To **train models that do not fit in GPU memory** without increasing the number of GPUs.

What is offloaded

- Optimizer states (e.g. Adam momentum & variance)
- Sometimes gradients (depending on configuration)
- Parameters remain logically the same (no model change)

What it is NOT

- Not a new parallelism strategy
- A **memory-capacity** technique, not a compute-scaling one

Number formats: how a float is stored

value = sign × 2^(exponent – bias) × 1.mantissa

More exponent bits → wider range. More mantissa bits → finer precision.



Format	Largest value	Next value after 1.0
FP32 32 bit	3.4×10^{38}	1.0000001
FP16 16 bit	65,504	1.001
BF16 16 bit	3.4×10^{38}	1.008
FP8 E4M3 8 bit	448	1.125
FP8 E5M2 8 bit	57,344	1.25
FP4 E2M1 4 bit	6	1.5

- **BF16** keeps FP32's range with fewer mantissa bits, so values rarely overflow. FP16 is more precise but overflows above 65,504, so it needed tricks such as loss scaling.
- **FP8** comes in two flavours: E4M3 (more precision) is often used for weights and activations, E5M2 (more range) for gradients.

Block scaling: making 8-bit and 4-bit work

FP4 has only 15 distinct values. One scale cannot fit all the numbers in a large tensor. **Fix: split the tensor into small blocks and give each block its own scale factor.**

Example: one block of 6 values, stored as FP4 (simplified NVFP4)

Original (BF16)	0.21	−0.05	0.62	1.30	−0.88	0.40
÷ scale (1.30 / 6 = 0.217)	0.97	−0.23	2.86	6.00	−4.06	1.85
Round to nearest FP4	**1**	**0**	**3**	**6**	**−4**	**2**
× scale (restored)	0.22	**0.00**	0.65	1.30	−0.87	0.43

- The largest value in the block maps to FP4's maximum (6)
- Only the FP4 codes and one scale per block are stored
- **Small values lose detail: −0.05 became 0**
- Blackwell and Rubin tensor cores apply the scales in hardware

Format	Element type	Block size	Scale per block	Bits per value
MXFP8	FP8 (E4M3 or E5M2)	32	8-bit power of two (E8M0)	8.25
MXFP4	FP4 (E2M1)	32	8-bit power of two (E8M0)	4.25
NVFP4	FP4 (E2M1)	16	FP8 E4M3, plus one FP32 scale per tensor	4.5

Sources: OCP Microscaling Formats (MX) Specification v1.0, 2023; NVIDIA Transformer Engine documentation (NVFP4), 2025.

Low-precision training: fewer bits per number

Format	Bytes	Role in training today	Hardware
FP32	4	Master weights and optimizer states (kept for accuracy)	All GPUs
BF16	2	Default for weights, activations and gradients in matmuls	A100 and later
FP8 (E4M3 / E5M2)	1	Matmuls in forward and backward; used at 671 B scale by DeepSeek-V3 (2024) [1]	H100 and later
MXFP8 / NVFP4	1 / 0.5	Block-scaled formats; NVFP4 pre-trained a 12 B model on 10 T tokens, matching FP8 (2025) [2]	Blackwell, Rubin

Why systems people care

Each halving of bits gives roughly:

- 2× tensor-core throughput
- ½ the memory for those tensors
- ½ the bytes sent over the network

What makes it work

- **Block scaling (previous slide):** DeepSeek-V3 did it in software on H100, with 1×128 tiles
- **Mixed precision:** keep sensitive parts in higher precision (master weights, optimizer, some layers)
- **FP4 needs more tricks:** random Hadamard transforms, stochastic rounding

[1] DeepSeek-AI, DeepSeek-V3 Technical Report, 2024. [2] NVIDIA, Pretraining Large Language Models with NVFP4, 2025.

Common Distributed Training Libraries

NVIDIA Megatron-LM

- Focus: scaling computation

Provides:

- Tensor (intra-layer) parallelism
- Pipeline parallelism
- Expert (MoE) parallelism

Optimized for:

- Large Transformer models

PyTorch FSDP

- Focus: scaling memory

Provides:

- Full sharding of parameters, gradients, optimizer states (ZeRO-3–style)

Characteristics:

- PyTorch-native API

TP / PP come from separate PyTorch modules, combined through DeviceMesh (FSDP2, TorchTitan)

DeepSpeed

- Focus: memory efficiency and flexibility

Provides:

- ZeRO (Stages 1–3)
- ZeRO-Offload / ZeRO-Infinity

Often used as:

- A memory backend combined with other parallelism frameworks

These libraries are complementary, not competing

- A common production setup:
  - Megatron-LM → model / pipeline / expert parallelism
  - FSDP or DeepSpeed → memory sharding and optimizer management

How to configure parallelism?

Automatic Parallelism (principled but difficult)

- **Goal**
  - Automatically choose the best parallelization strategy.
- **How it works**
  - Build a **cost model** (compute, memory, communication)
  - Search the space of parallel strategies to **minimize total cost**
- **Why it's hard**
  - Accurate cost modeling is challenging
  - Large search space
  - Hardware- and workload-dependent behavior

Empirical Parallelism (practical and widely used)

- **Key idea**
  - Parallelism choices are **model-aware and user-guided**
- **How it works**
  - Manually partition a small number of key operators
  - Automatically propagate partitioning to dependent operators
- **Why it works**
  - Leverages known model structure (e.g., Transformers)
  - Simple, predictable, and robust
- **In practice**
  - Used by **Megatron-LM, FSDP, DeepSpeed**

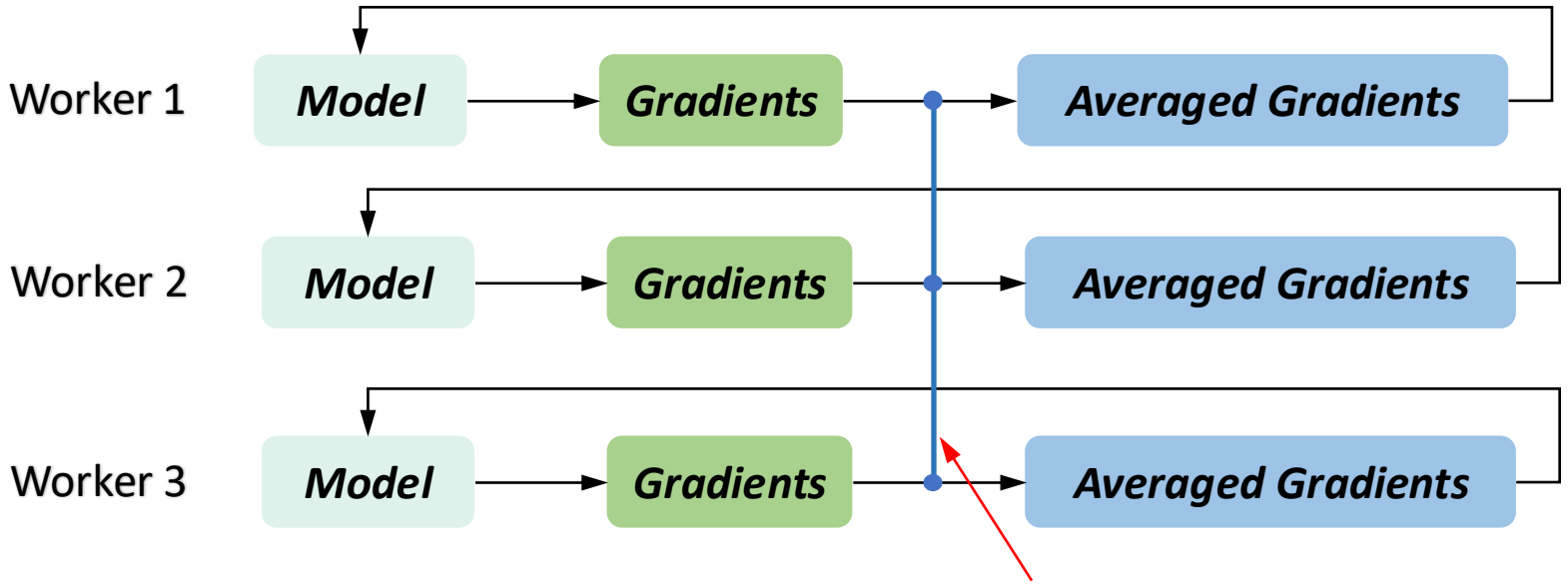
Empirical guide – suggested by Megatron Team [1]

- **Memory Optimization**
  - Use **Zero optimizer** to reduce optimizer state memory
  - Enable **sequence parallelism** when using tensor parallelism to reduce activation memory
  - Use **context parallelism** for long sequence training
  - Consider **pipeline parallelism** for very large models that don't fit on a single GPU

- **Performance Optimization**
  - **Tensor parallelism** works best within one NVLink domain (a server, or a whole NVL72 rack)
  - **Pipeline parallelism** can work across nodes but requires careful batch size tuning
  - **Context parallelism** is essential for long context scenarios
  - **Expert parallelism** is specific to MoE models and should match hardware

[1] https://docs.nvidia.com/nemo/megatron-bridge/0.2.0/parallelisms.html

Communication dominates parallelism performance



GPT-3: 175 B parameters

BF16 gradients: 175 B × 2 bytes = 350 GB per step

Ring allreduce: each GPU sends ≈ 2 × 350 GB = 700 GB

At 50 GB/s per GPU (400 Gb/s network): ≈ 14 s per step

This can exceed the compute time of the step!

Compute averaged gradients

1. Collect local gradients
2. Broadcast averaged gradients to all workers

GPU Server Architecture

Example: an 8×A100 server

- Memory hierarchy
  - GPU memory (HBM2e): ~2,000 GB/s per GPU
  - CPU memory (DDR4, 2 sockets): ~400 GB/s
  - Local NVMe SSDs: ~20 GB/s
- Interconnects
  - GPU–GPU: NVLink via NVSwitch, 600 GB/s per GPU
  - CPU–GPU: PCIe Gen4 x16, 32 GB/s each way
  - Server–server: InfiniBand HDR, 25 GB/s per NIC
- ML data placement: how to improve data locality and reduce access latency?

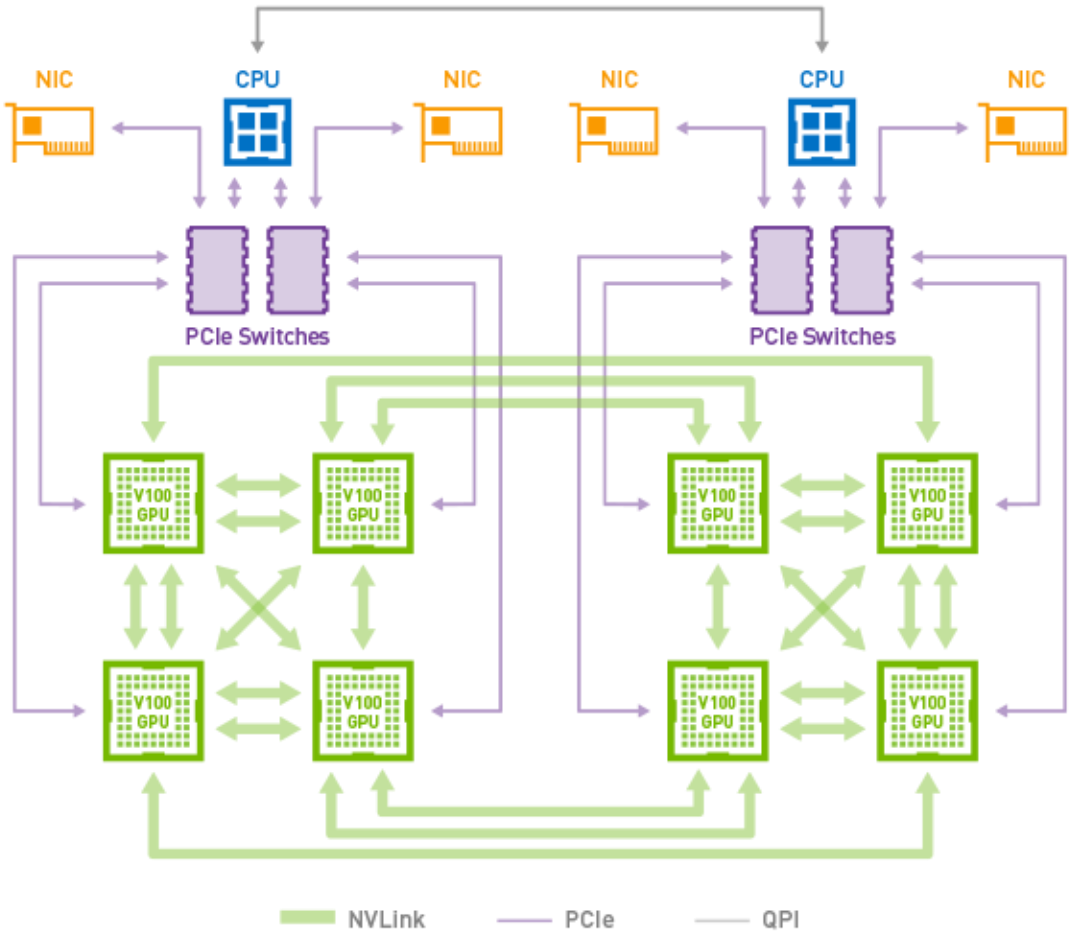
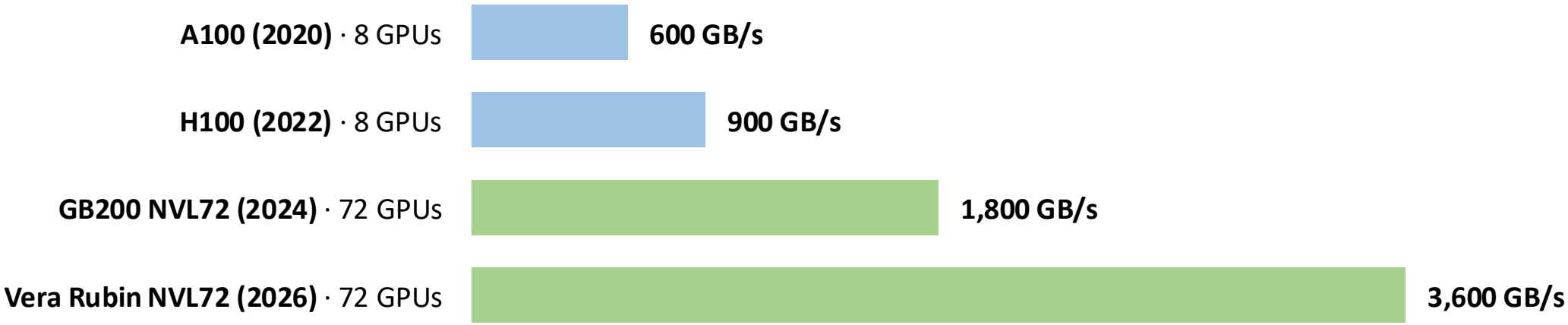


Figure: an older 8×V100 server (DGX-1) with a partial NVLink mesh. From A100 onwards, NVSwitch connects all 8 GPUs, so every pair gets full NVLink bandwidth.

Scale-up domains keep growing

NVLink bandwidth per GPU (bidirectional) and number of GPUs in one NVLink domain



Scale-up vs scale-out

- Network between domains: 25 → 50 → 100 GB/s per GPU (200 → 400 → 800 Gb/s NICs)
- **NVLink inside a domain is about 20× faster**

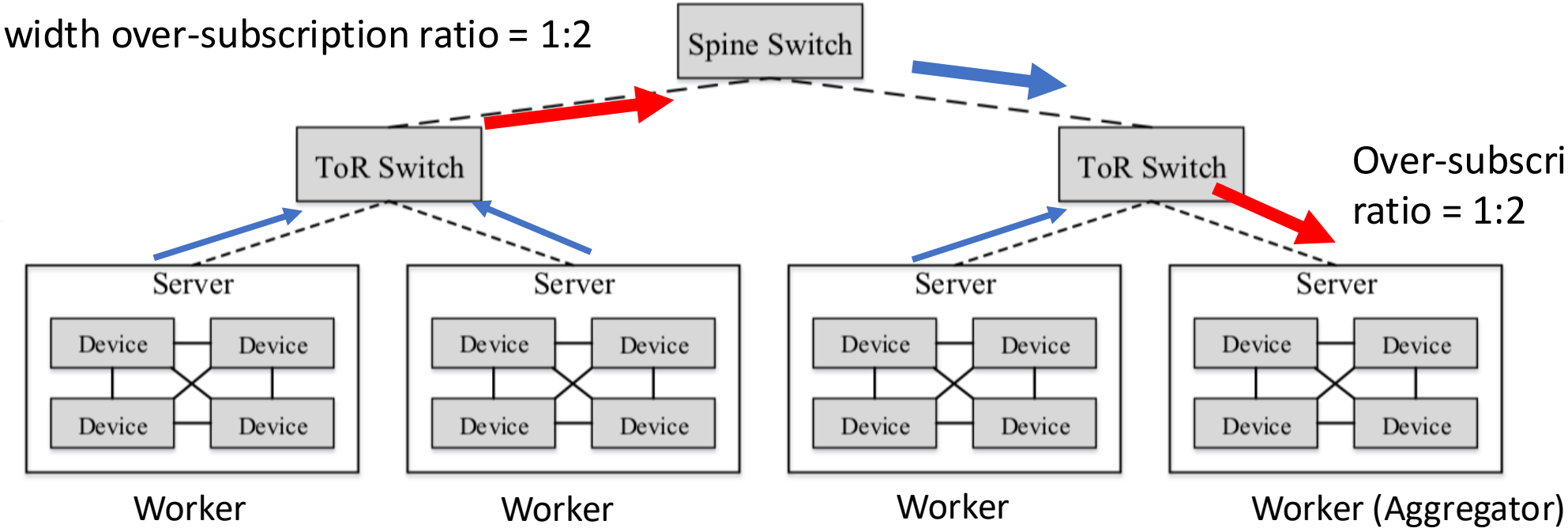
What this means for parallelism

- Put the heaviest communication (TP, EP, CP) inside the NVLink domain
- Use DP / FSDP / PP across domains
- With NVL72, the domain is a whole rack, not one server

Sources: NVIDIA product specifications. Vera Rubin NVL72 was announced at CES in January 2026 for the second half of 2026.

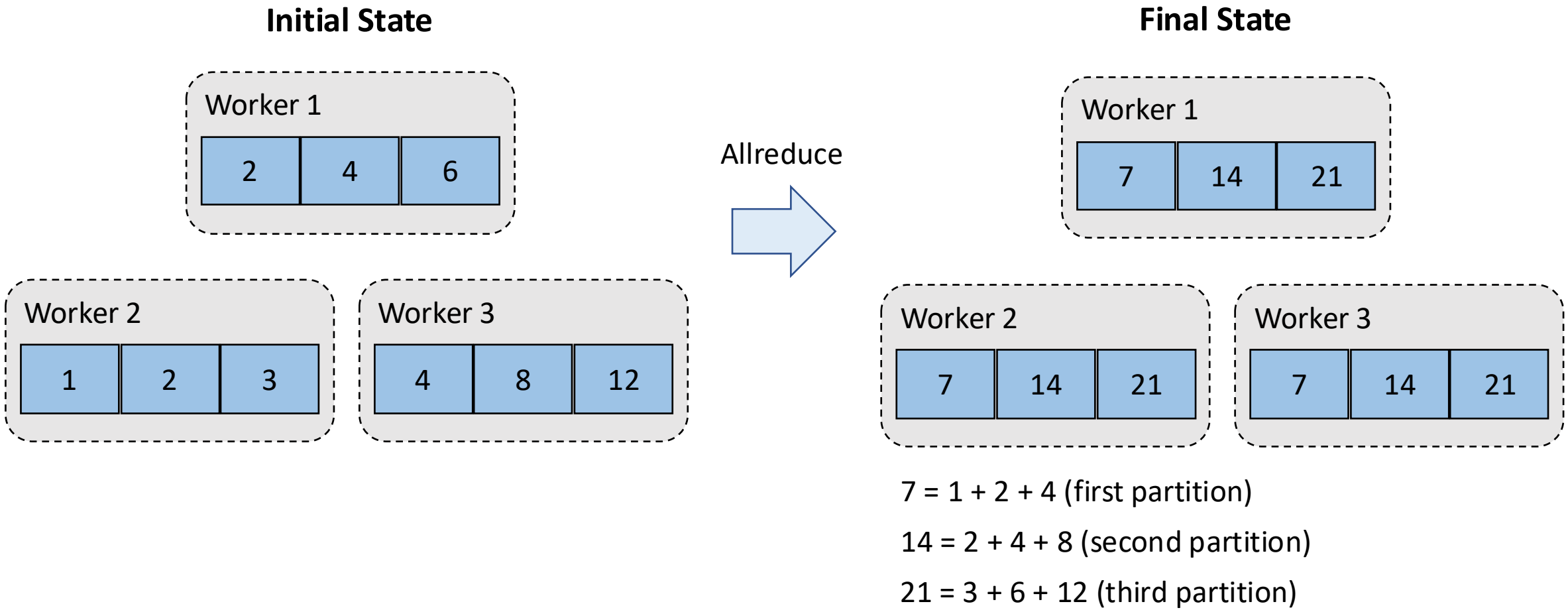
Bandwidth is Limited in Data Centres

Bandwidth over-subscription ratio = 1:2



A real-world data centre

Bandwidth-efficient Allreduce

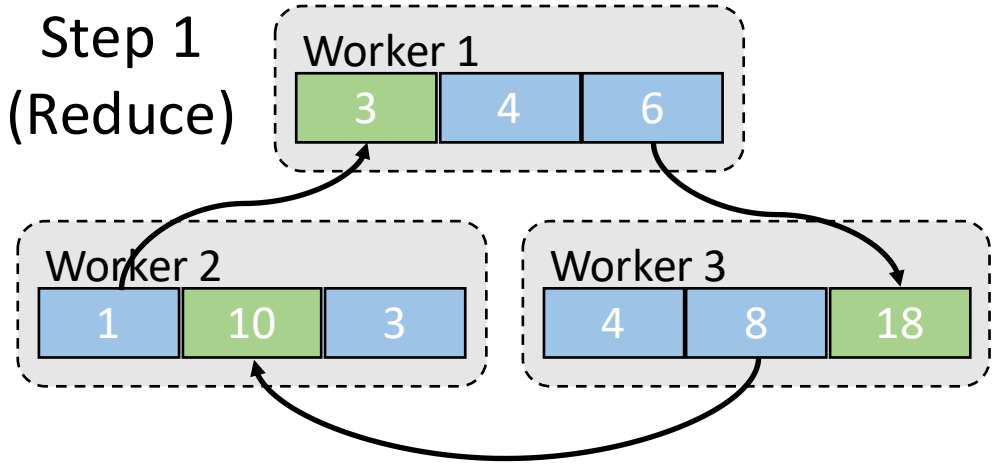


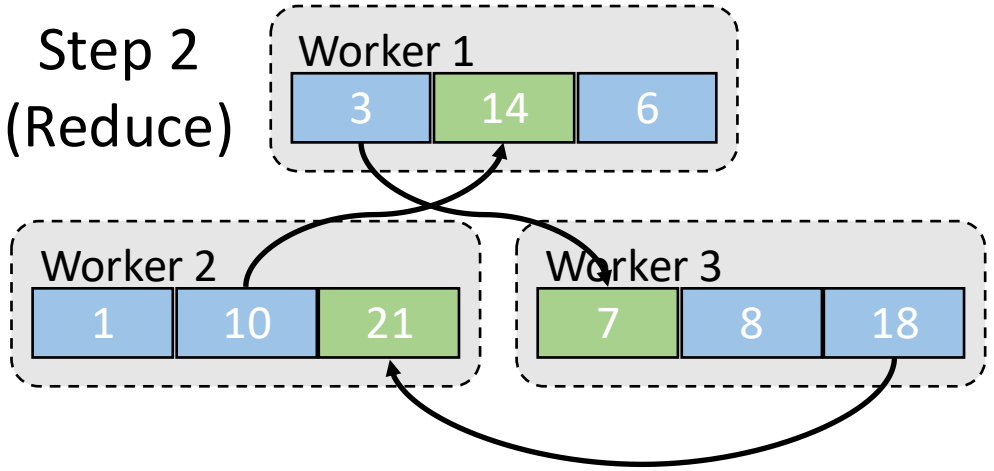
7 = 1 + 2 + 4 (first partition)

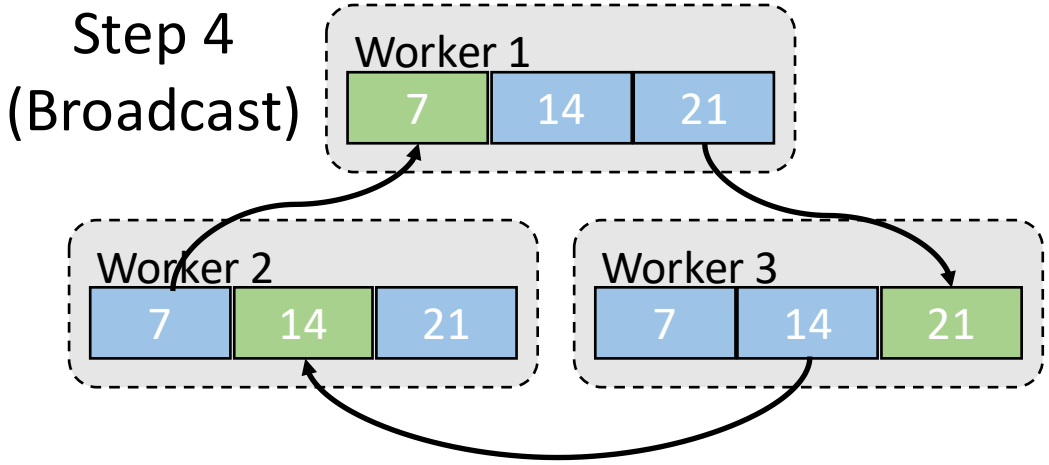
14 = 2 + 4 + 8 (second partition)

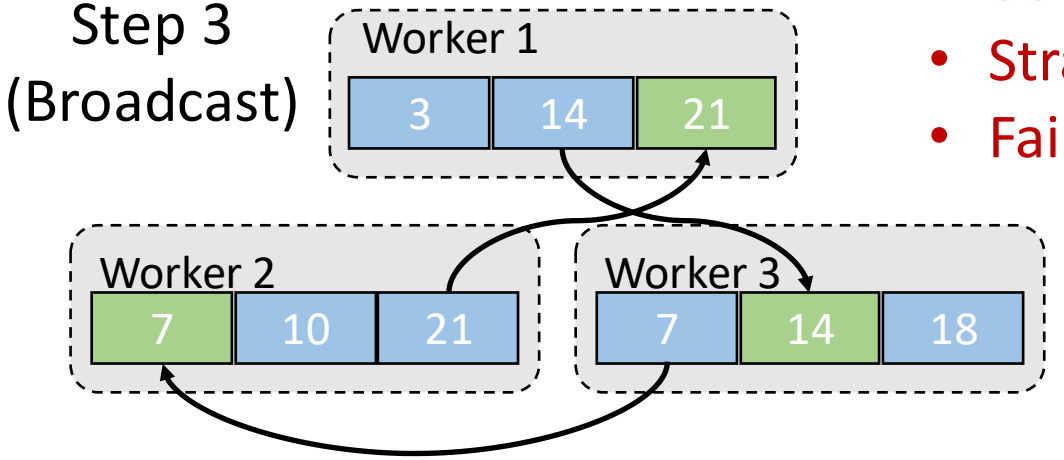
21 = 3 + 6 + 12 (third partition)

Steps in Allreduce





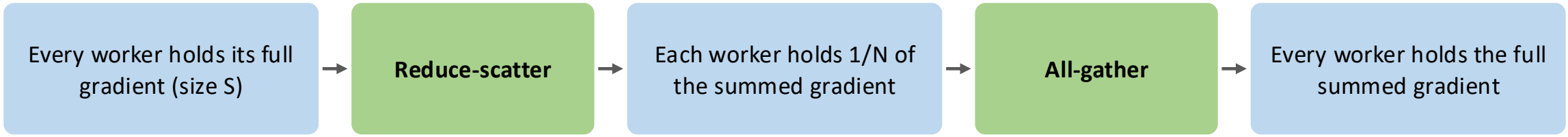




Discussion

- Stragglers
- Failures

Ring allreduce = reduce-scatter + all-gather



Steps 1–2 on the previous slide

Steps 3–4 on the previous slide

Cost per worker

Data sent = $2(N-1)/N \times S \approx 2S$

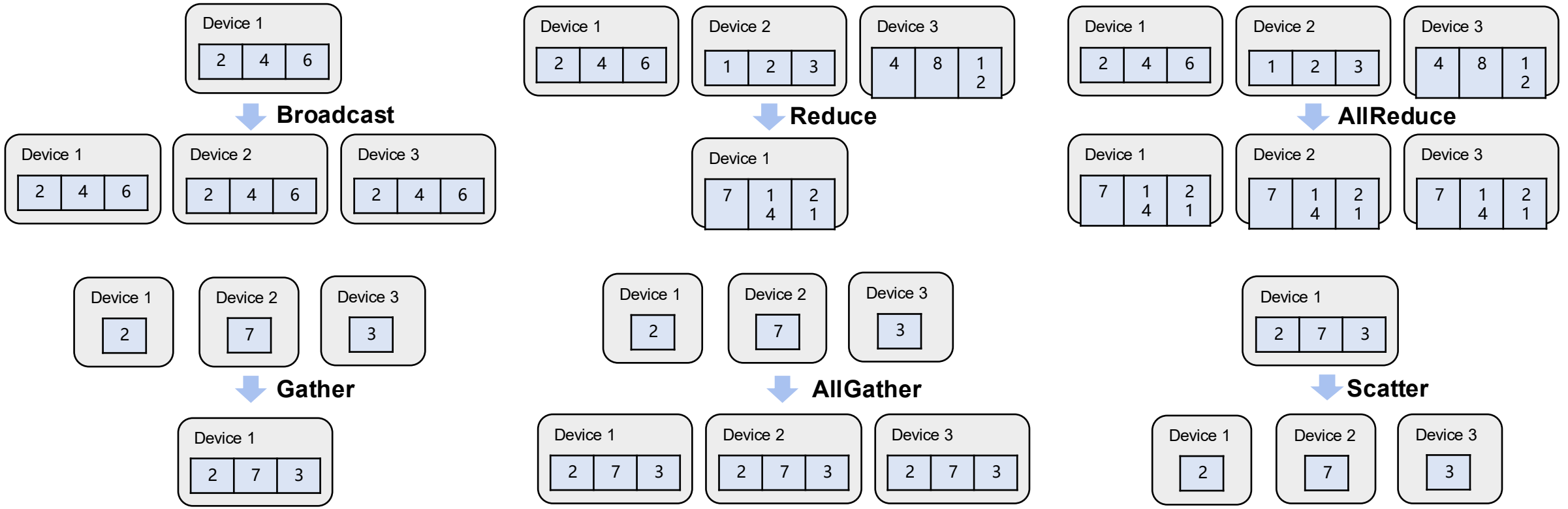
- Almost independent of N → bandwidth-optimal
- But it takes $2(N-1)$ steps, so latency grows with N. Large clusters use tree or hierarchical algorithms (NCCL chooses automatically).

The same pieces reappear in ZeRO / FSDP

- ZeRO-2: keep only the reduce-scatter; each GPU updates its own shard
- ZeRO-3 / FSDP: all-gather each layer's parameters just before use, then free them
- **Same collectives, rearranged in time**

Overlap: start the allreduce of layer L's gradients while backward is still computing layer L – 1 (gradient bucketing in PyTorch DDP).

Common collective communication operators

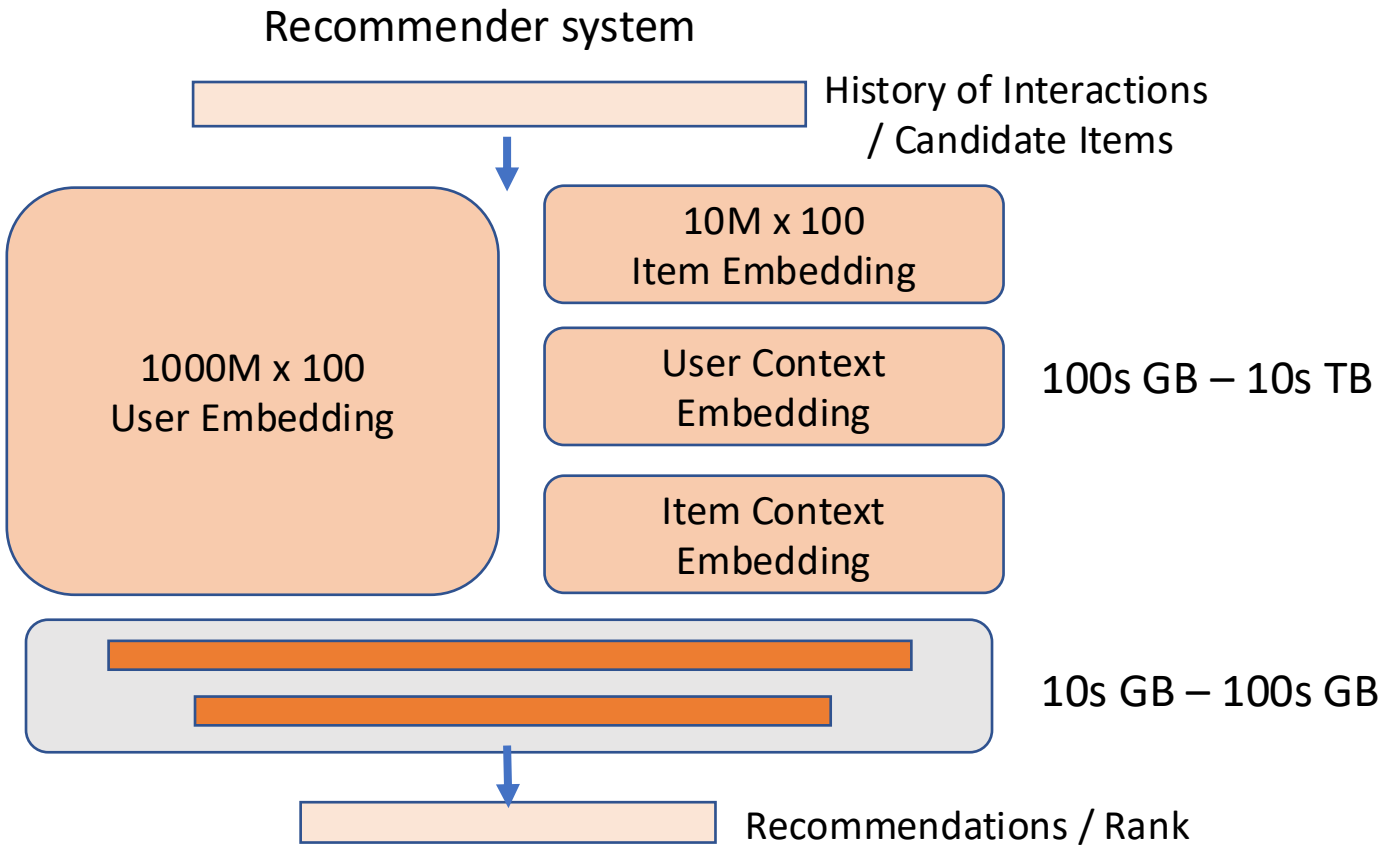


- Collective communication libraries: NVIDIA Collective Communications Library (NCCL), OpenMPI
- Integration with ML frameworks: PyTorch Distributed, Horovod, KungFu

Topologies for AI clusters

FatTree	All-to-All	Mesh	Torus
What it is • Hierarchical switch-based network • Bandwidth increases toward the root **Pros** • Scales to thousands of GPUs • Matches data center realities **Cons** • Oversubscription is common Expensive at scale **Example** InfiniBand / Ethernet clusters	**What it is** • Every GPU connects directly to every other GPU **Pros** • Lowest latency • Ideal for collective-heavy workloads (e.g., AllReduce) **Cons** • **Does not scale** ($O(N^2)$ links) • Extremely expensive and power-hungry **Example** • NVLink + NVSwitch domains (8 GPUs per server; 72 per NVL72 rack)	**What it is** • GPUs arranged in a 2D grid • Each GPU connects to neighbors **Pros** • Scales well • Simple routing • Hardware-friendly **Cons** • Communication latency depends on distance • AllReduce becomes topology-aware **Example** • Wafer-scale accelerators, some custom AI chips	**What it is** • Mesh + wrap-around links **Pros** • Shorter average path length • Better bandwidth utilization than mesh **Cons** • More complex routing • Still not fully connected **Example** • Classic HPC systems, TPU-style designs

Practical System 1: Deep Learning Recommender Systems

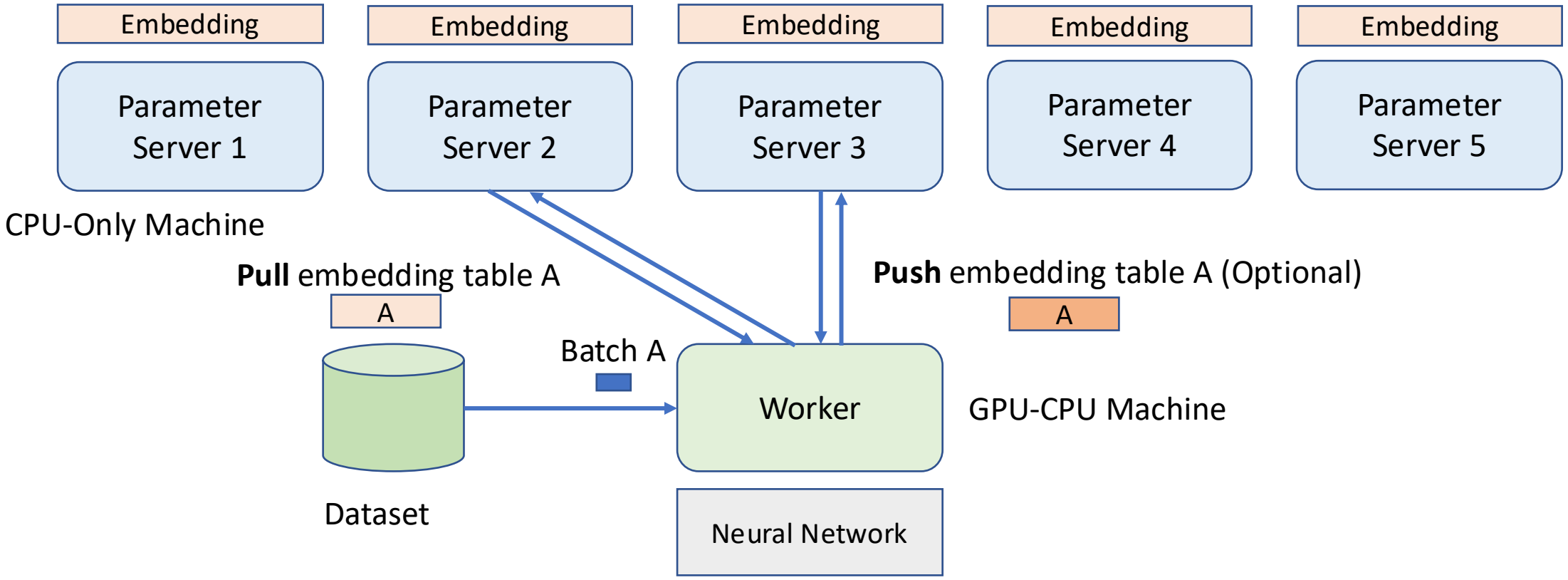


- Tremendous memory cost
- Small computation cost
- Embeddings are **sparsely updated**

- Example: Deep learning recommendation models [1]

[1] DLRM: An advanced, open source deep learning recommendation model, 2020

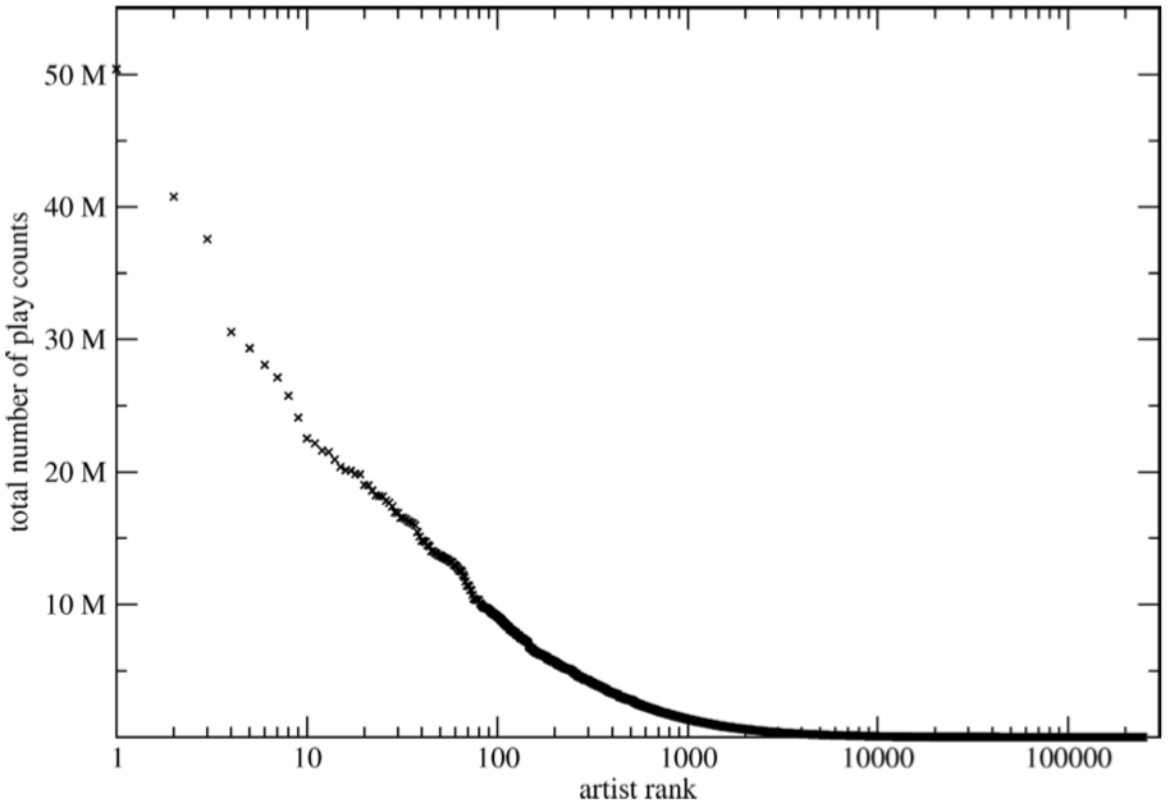
Parameter Server Architecture



Parameter Servers vs. Collectives: When to Use Which?

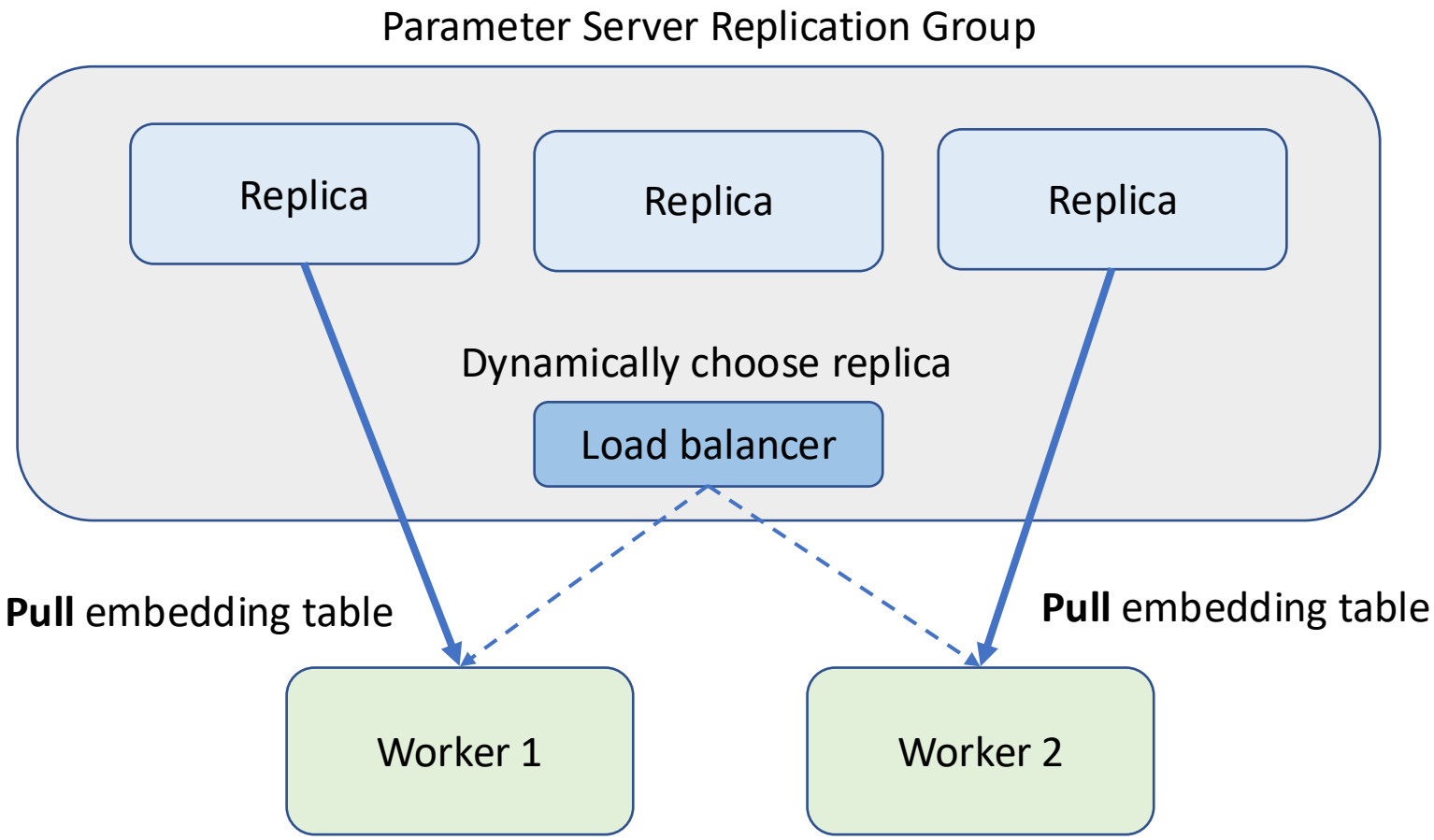
1. Use parameter servers when updates are asynchronous or compute-light;
2. Use collective communication when training is synchronous and compute-heavy.

Users/Items Follow Power Law Distribution



Imbalanced workload is common in distributed computing systems

Handling Hot Spots via Data Replication

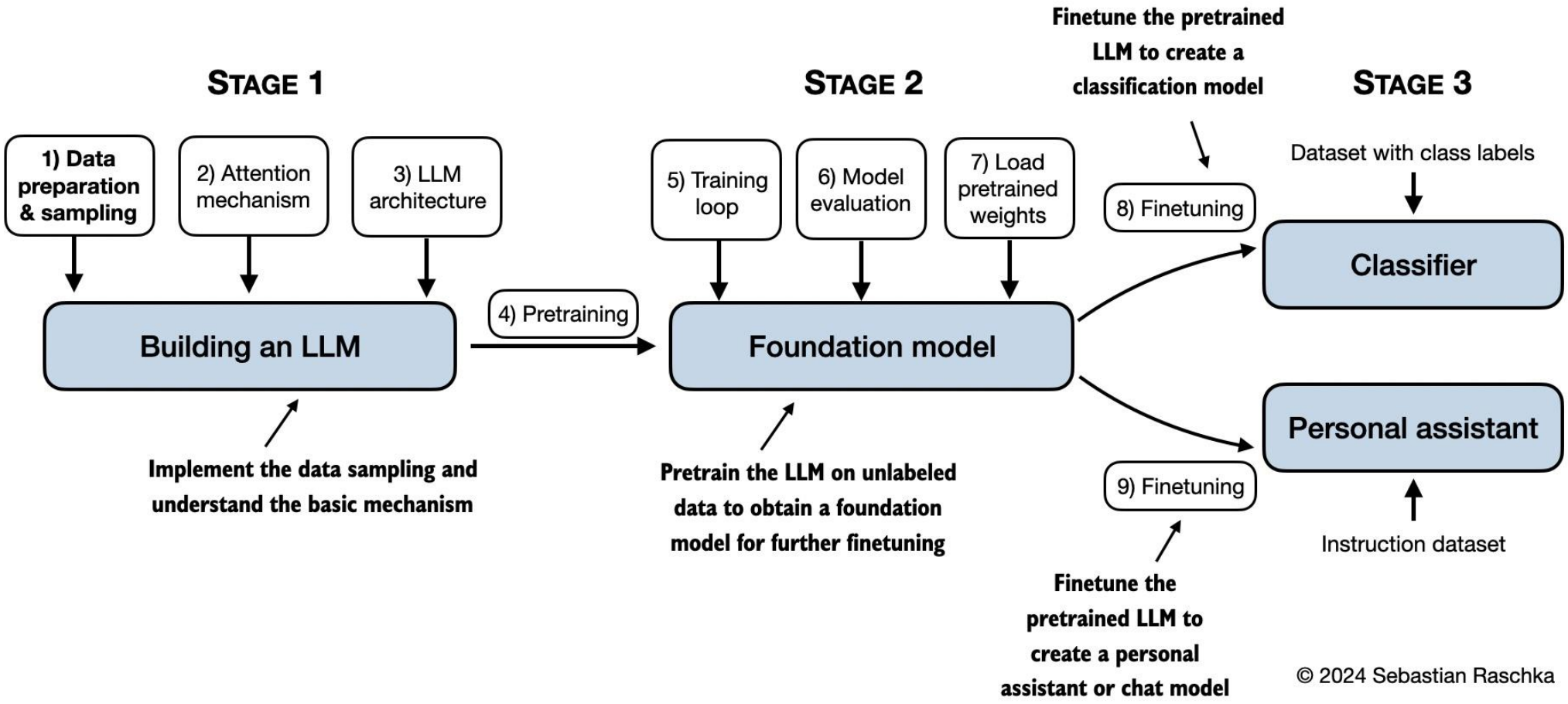


Some practical concerns

- Sensitive user data – federated learning [1]
- Privacy-aware training
- Continuous model updates [2]
- Geo-distributed replicas
- Data freshness vs stability
- Tight latency budgets
- Online evaluation

[1] Towards Federated Learning at Scale: System Design, MLSys 2019
[2] Ekko: A Large-Scale Deep Learning Recommender System with Low-Latency Model Update, OSDI 2022

Practical System 2: LLM training systems



[1] https://github.com/rasbt/LLMs-from-scratch

Pre-training

Simple intuition:
Pre-training teaches the model how language works.

It learns:
- Words
- Grammar
- Facts
- patterns

Without being told what to do with them.

Very simple example

*Imagine you give the model **millions of sentences** like:*

"The cat sat on the ___."
"Paris is the capital of ___."
"If it rains, bring an ___."

*You ask **only one question**:*

"What word comes next?"

That's pre-training.
- *No labels like positive/negative*
- *No task instructions*
- *Just predicting missing or next words*

Failures are the norm at scale

419

unexpected job interruptions in 54 days of Llama 3 pre-training on 16,384 H100 GPUs [1]

~3 hours

average time between interruptions

78%

caused by hardware, mostly faulty GPUs and HBM memory

How training systems respond

- **Checkpoint often and cheaply.** Copy state to CPU memory fast, then write it to storage in the background. Keep in-memory copies on peer nodes for a quick restore.
- **Detect problems quickly.** Health checks and communication timeouts. Watch for stragglers: in synchronous training, one slow GPU slows everyone.
- **Recover quickly.** Swap in spare nodes and restart only what failed.
- **Adapt the job.** Elastic training changes the number of GPUs and re-shards model state without starting over (e.g., Tenplex [2]).

Result: Llama 3 still achieved more than 90% effective training time.

[1] Grattafiori et al., The Llama 3 Herd of Models, 2024. [2] Wagenländer et al., Tenplex: Dynamic Parallelism for Deep Learning using Parallelizable Tensor Collections, SOSP 2024.

Post-training (Supervised Fine-tuning)

Simple intuition:
Post-training teaches the model how to use its knowledge, and what is good or bad

Now we tell it:
- what the task is
- what a “good answer” looks like

Example: Personal assistant / chat

You show the model:

Input

“Explain gradient descent to a beginner.”

Target output

“Gradient descent is a method that…”

Now the model learns:
- *follow instructions*
- *be helpful*
- *format answers nicely*

Post-Training (Reinforcement Learning)

Simple intuition:
Reinforcement Learning teaches the model by letting it try, fail, and improve.

Instead of being told the correct answer:

- the model **acts**
- the system/environment **reacts**
- the model receives a **reward**
- the model adjusts its behavior

Reinforcement Learning example (core idea)

You give the model a prompt:

"Summarize this email."

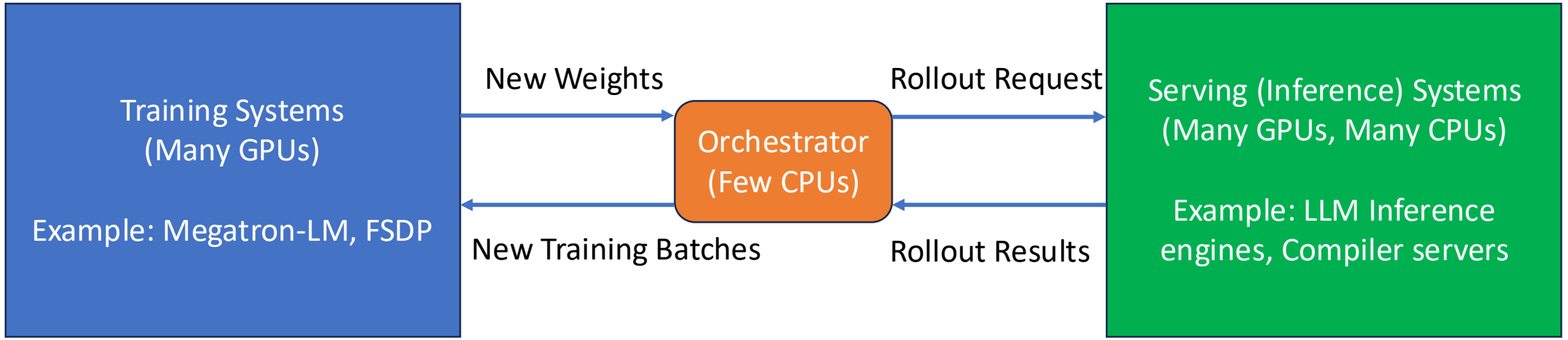
The model generates **an answer** on its own.

Now the system does **not** give a correct answer.
Instead, it gives a **reward** via a rollout process**:**

- +1 if the summary is helpful
- 0 if it's vague
- −1 if it's wrong or unsafe

The model does **not** know *why* — only **how good it was**.

Advanced Topic: Emerging RL-based Training System



- New weights are often big and can cause a transfer bottleneck
  - Support shared checkpoints for training and inference (see veRL library)
- Many rollout requests run in parallel, with some taking much longer than others
  - Synchronous RL vs. Asynchronous RL

Scaling Law – what is next?

Pre-training scaling: reaching its limits

- More parameters + more data → **smaller gains per dollar**
- High-quality data is **running out**
- Training costs, energy, and time are **growing super-linearly**

Emerging AI scaling paradigms:

- **Inference / test-time scaling:** More compute per query → better reasoning (search, CoT)
- **Post-training scaling:** SFT, RL, preference optimization, tool use
- **Data scaling (quality > quantity):** Curated, synthetic, and feedback-driven data

All AI scaling paths reduce to the same system fundamentals: parallelism, batching, memory hierarchy, and compiler/runtime efficiency.

Emerging Challenges for Training Systems

- **Extreme activation memory pressure** (long context, multimodality)
- **Communication is becoming the bottleneck**, not compute
- **Heterogeneous hardware** (GPUs, NPUs, interconnects)
- **Composing many parallelisms** reliably at scale
- **Dynamic workloads** (MoE routing, variable sequence length)
- **Training–serving co-design** (lifetime continual learning)
- **Reliability at scale** (failures and stragglers across 10K–100K GPUs)
- **Low-precision numerics** (FP8 / FP4 training without losing accuracy)

Design principles of training systems

Five ideas that connect everything in this lecture

1. **Find the bottleneck first**
   Is it memory, compute or communication? Measure MFU and memory before optimising.

2. **Trade one resource for another**
   Recompute to save memory (checkpointing), communicate to save memory (ZeRO), use fewer bits to save everything (FP8 / FP4).

3. **Match parallelism to the bandwidth hierarchy**
   TP / EP / CP inside the NVLink domain; DP / FSDP / PP across servers and racks.

4. **Overlap communication with computation**
   Gradient bucketing, FSDP prefetching, better pipeline schedules, DualPipe.

5. **Design for failure and imbalance**
   Checkpointing and elastic training; balancing hot experts, hot embeddings and slow RL rollouts.